A robust algorithm to find the point closest to a collection of skew lines and the line closest to a collection of skew points.

Daniel Karron

January 19, 1992

This technical report presents an algorithm and code to solve for two similar classes of problems. Given a list of points, find the equations for the best line that passes a minimum distance from each point. Given a list of lines, find the point that is a minimum distance from each line. Singular value decomposition is used to solve for the inconsistant matrix. Numerical robustness is tested for alternate solutions to the line problem, and the best solution form is returned. This avoids singularities in best fit lines that are oriented along certain directions. This algorithm is implemented in "C", and has applications in locating points in tomographic slice stacks, and finding trajectories of high energy particles passing through arrays of detectors.

The source code is presented in the appendix.

- Find the line that has the minimum RMS[1] distance from a list of points denoted (x_i, y_i, z_i), $i = 1, 2, 3, \ldots$
- Find that point that has the mimimum RMS distance from a given list of lines.

*This work was supported by an equipment loan from Silicon Graphics, Inc. D. Karron is with Department of Applied Science at New York University and the Department of Surgery at New York University Medical Center, 560 First Avenue, New York, New York, 10016.

[1]Root Mean Squared

Abstract

An algorithm is presented to solve for two similar classes of problems. Given a list of points, find the equations for the best line that passes a minimum distance from each point. Given a list of lines, find the point that is a minimum distance from each line. Singular value decomposition is used to solve for the inconsistant matrix. Numerical robustness is tested for alternate solutions to the line problem, and the best solution form is returned. This avoids singularities in best fit lines that are oriented along certain directions. This algorithm is implemented in "C", and has applications in locating points in tomographic slice stacks, and finding trajectories of high energy particles passing through arrays of detectors.

1

It is crucial to represent each line as the intersection of two planes. The solution of the second problem then follows simply from the results of the first problem. Each line is given as a pair of intersecting plane equations :

$$A\ x_i + B\ y_i + C\ z_i = 1$$

$$D\ x_i + E\ y_i + F\ z_i = 1$$

where $i = 1, 2, 3, \ldots, n$

The first problem can be formulated as

$$[A]\ [X] = [B]$$

where

$$[A] = \begin{bmatrix} x_0 & y_0 & z_0 \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \vdots & \vdots & \vdots \\ x_n & y_n & z_n \end{bmatrix}$$

$$[X] = \begin{bmatrix} A & D \\ B & E \\ C & F \end{bmatrix}$$

$$[B] = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ \vdots & \vdots \\ 1 & 1 \end{bmatrix}$$

Note that the data matrix $[A]$ has, in general, many more rows than columns, so that this system is usually over determined. The SVD (Singular Value Decomposition) solution is thus one of many possible solutions and can be shown to be the one with the required RMS error properties.

The SVD technique is based on the observation that any matrix $[A]$ whose number of rows N is greater or equal to its number of columns N , can be

written as the product of an $N \times M$ c lumn orthogonal matrix [U], an $M \times M$ diagonal matrix [W], with positive orj zero elements, and the transpose of an $M \times M$ orthogonal matrix *[V] [1], [2],* 3]; that is,

$$[A] = [dI[W][V]T.$$

The "standard" SVD solution, which can be verified by back substitution, is given as:

$$[X] = [V][W]-1[U]T\ [B]$$

The problem with this solution is that the pairs of planes that result can be close to parallel, leading to severe numerical instabilities if these values are used to solve the second problem. To minimize these numerical instabilities we constrain the solution planes to be perpendicular. Any calculation with these lines will suffer from numerical instability if the rows of the line matrix are not "maximally" linearly independent.

Intuatively, think of the line as two intersecting planes. Think of each plane as minimally defined as three points. Now at first glance it would appear that 6 points are required to define two planes. Remember that the planes can have points in common, and that we can take two triangles for two planes that share two points. So we can minimally solve for two planes in space with 4 points. If we now constrain our planes to be perpendicular, then we do not need one point. Now if we also allow our perpendicular planes to rotate about the line, then we do not need the remaining third point to constrain the perpendicular planes in any orientation with respect to our line. Therefore, we can use two points to define a line, but the orientations of the planes are undefined, but perpendicular to each other.

From the above argument, it is clear that there are multiple equivalent pairs of planar equations for any line. Perpendicularity means that any one coefficient from the first plane $\{A, B, C\}$ and any one coefficient not in the same column $\{D, E, F\}$ can be 0. This is equivalent to projecting the line onto any two of the $x — y, y — z,$ or z — x planes. This reduces the 2 x n overdetermined equations in 6 unknowns to a three systems of n overdetermined equations in two unknowns. However, if any of the x; ,y; , or z; values for our lines approach zero, then one pair of equations becomes singular. To avoid singularity in any particular direction, we must determine which three pairs of equations to use. Since each

```
   {
   b[i]=1.0; /* point normal plane constant */
   for(j=1;j≤2;j++)
      {
      u[i][j]=a[i][j]; /* don't touch matrix a */
      }
   }

#ifdef DEBUG
/* Lets see the intput */
Print_Matrix("a          ",1,a,1,m,1,n);
Print_Vector("constants",1,b,1,m);
#endif DEBUG

svdcmp(u,m,n,w,v); /* decompose u into u,v, and w */

wmax=0.0; /* edit diagnonal which is vector v */
for(i=1;i≤m;i++)
   {
   if(w[i] > wmax)
      {
      wmax=w[i];
      }
   }

#ifdef DEBUG
printf("wmax=%f\n",wmax);
#endif DEBUG

wmin=wmax*(1.0e-6);
for(i=1;i≤m;i++)
   {
   if(w[i] < wmin)
      {
      w[i]=0.0;
      }
   }

#ifdef DEBUG
printf("wm          n",wmin);
#endif DEBUG


svbksb(u,w,v,m,n,b,x); /* svd back substitute to solve for best x */
```

```
#ifdef DEBUG
Print_Matrix("lhs",1,u,1,m,1,n);
Print_Vector("d"          ,1,w,1,m);
Print_Matrix("rhs",1,v,1,m,1,n);
Print_Vector("solut"      ,1,x,1,n);
#endif DEBUG

nrdot(a,1,m,1,n,x,c);

#ifdef DEBUG
Print_Vector("test solut"          )",1,c,1,m);
#endif DEBUG

condition=0.0; /* this is not really condition, just an figure of merit
            * for the absolute error, as absolute difference from 1.0
            */

for(i=1;i≤m;i++)
   {
   if((c[i]−1.0)<0.0)
      condition −= (c[i]−1.0);
   else
      condition += (c[i]−1.0);
   }

#ifdef DEBUG
printf("cond"    n=%f\n",condition);
#endif DEBUG

free_matrix(u,1,m,1,n);
free_matrix(v,1,m,1,n);
free_vector(b,1,m);
free_vector(c,1,m);
free_vector(w,1,n);

return condition;
}
/*****************************************************************************/
void NueNogginKnocker(int m,float **plist,float lcoeff[2][3])
{
/*
 * Given a list of m point triples (x,y,z) (which actually should be declared
 * (*p)[3] but I had trouble making that work)  return the point normal line
```

```
 * coefficients (where the point constant is taken as 1.0, always!)
 * in lcoeff for two point normal planes that describe the line that
 * best fits the list of points in three dimensions.
 */

register int i;
float **a,*x[4];
double c[4];
int cmaxi;
double cmaxf;
static int n=2;

#ifdef DEBUG
Debug=2;
#endif DEBUG

#ifdef DEBUG
for(i=0;i<m;i++) /* Recite the incomming point list just to be certain. */
   {
   printf("NueNogg   nocker Po              ,%f,%f\n",i,
      *(plist[i]), *(plist[i]+1), *(plist[i]+2));
   }
#endif DEBUG

if( m < n )
   {
   fprintf(stderr,"too few po     %d<%d to do anyth    M<N\n",m,n);
   return;
   }

a=matrix(1,m,1,n);
x[1]=vector(1,n); /* hang three solution vectors off x */
x[2]=vector(1,n);
x[3]=vector(1,n);

/* first try */
for(i=0;i<m;i++)
   {
   a[i+1][1]= *(plist[i]);  /* x */
   a[i+1][2]= *(plist[i]+1);/* y */
   }

#ifdef DEBUG
printf("f      try\n");                                                    rs
```

```
#endif DEBUG

c[1]=RunCalculation(a,m,x[1]);

/* second try */

for(i=0;i<m;i++)
   {
   a[i+1][1]= *(plist[i]);  /* x */
   a[i+1][2]= *(plist[i]+2);/* z */
   }
#ifdef DEBUG
printf("second try\n");
#endif DEBUG

c[2]=RunCalculation(a,m,x[2]);

/* third try */

for(i=0;i<m;i++)
   {
   a[i+1][1]= *(plist[i]+1); /* y */
   a[i+1][2]= *(plist[i]+2); /* z */
   }

#ifdef DEBUG
printf("th[illegible]  try\n");
#endif DEBUG

c[3]=RunCalculation(a,m,x[3]);

cmaxf=0.0; /* which is the worst of the three ?, keep the two best ! */
for(i=1;i≤3;i++)
   {
   if(cmaxf<c[i])
      {
      cmaxf=c[i];
      cmaxi=i;
      }
   }

#ifdef DEBUG
printf("cmax=%f,cmax[illegible]=%d\n",cmaxf,cmaxi);
#endif DEBUG
```

```
switch(cmaxi) /* wire the best two solutions
               * back into the solution for the line
               */
    {
    case 3:
    lcoeff[0][0]=x[1][1];
    lcoeff[0][1]=x[1][2];
    lcoeff[0][2]=0.0;

    lcoeff[1][0]=x[2][1];
    lcoeff[1][1]=0.0;
    lcoeff[1][2]=x[2][2];
    break;

    case 2:

    lcoeff[0][0]=x[1][1];
    lcoeff[0][1]=x[1][2];
    lcoeff[0][2]=0.0;

    lcoeff[1][0]=0.0;
    lcoeff[1][1]=x[3][1];
    lcoeff[1][2]=x[3][2];
    break;

    case 1:
    lcoeff[0][0]=x[2][1];
    lcoeff[0][1]=0.0;
    lcoeff[0][2]=x[2][2];

    lcoeff[1][0]=0.0;
    lcoeff[1][1]=x[3][1];
    lcoeff[1][2]=x[3][2];
    break;
    }

#ifdef DEBUG
printf("lcoeff[0]= %f,%f,%f\n",lcoeff[0][0],lcoeff[0][1],lcoeff[0][2]);
printf("lcoeff[1]= %f,%f,%f\n",lcoeff[1][0],lcoeff[1][1],lcoeff[1][2]);
#endif DEBUG

free_matrix(a,1,m,1,n);
free_vector(x[1],1,n);
```

```
free_vector(x[2],1,n);
free_vector(x[3],1,n);
}
/*****************************************************************************/
#ifdef OBSOLETE
/* this version does not check for best two solutions out of three,
 * and has a bad numerical instability in certain directions.
 */

void NogginKnocker(int m,float **plist,float lcoeff[2][3])
{/* project onto two planes, first pass= x,y */
/*  second pass, x,z */
register int i,j,k;
float **u,*w,**v,**a,*b,*c,*x,**cvm;
float wmax,wmin;
int n=2;
float a1,a2,b1,c2;

#ifdef DEBUG
Debug=0;
#endif DEBUG

for(j=0;j<m;j++)
   {
   printf("Po%          %f,%f,%f\n",j,
      *(plist[j]), *(plist[j]+1), *(plist[j]+2));
   }

if(m < n )
   {
   fprintf(stderr,"Umlout M<N\n");
   return;
   }

a=matrix(1,m,1,n);
b=vector(1,m);
c=vector(1,m);
u=matrix(1,m,1,n);
v=matrix(1,m,1,n);
cvm=matrix(1,m,1,n);
w=vector(1,m);
x=vector(1,n);

for(i=0;i<m;i++)
```

```
   {
   b[i+1]=1.0;

   u[i+1][1]=a[i+1][1]= *(plist[i]); /* x */
   u[i+1][2]=a[i+1][2]= *(plist[i]+1);/* y */
   }

Print_Matrix("F%d    Pass, %d      ,1,a,1,m,1,n);
Print_Vector("F%d    Pass, constants",1,b,1,m);
svdcmp(u,m,n,w,v);
wmax=0.0;
for(i=1;i≤m;i++)
   {
   if(w[i] > wmax)
      {
      wmax=w[i];
      }
   }
#ifdef DEBUG
printf("F%d    Pass wmax=%f\n",wmax);                                    rs
#endif DEBUG
wmin=wmax*(1.0e-6);
for(i=1;i≤m;i++)
   {
   if(w[i] < wmin)
      {
      w[i]=0.0;
      }
   }
svbksb(u,w,v,m,n,b,x);
#ifdef DEBUG
Print_Matrix("lhs",1,u,1,m,1,n);
Print_Vector("d%d          ,1,w,1,m);
Print_Matrix("rhs",1,v,1,m,1,n);
Print_Vector("F%d    pass solut%d     ,1,x,1,n);
nrdot(a,1,m,1,n,x,c);
Print_Vector("F%d    pass test solut%d         ?)",1,c,1,m);
#endif DEBUG

a1=x[1];
b1=x[2];

/* second pass */
```

```
for(i=0;i<m;i++)
   {
   u[i+1][1]=a[i+1][1]= *(plist[i]); /* x */
   u[i+1][2]=a[i+1][2]= *(plist[i]+2);/* z */
   }

Print_Matrix("Second Pass, ",1,a,1,m,1,n);
Print_Vector("Second Pass, constants",1,b,1,m);
svdcmp(u,m,n,w,v);
wmax=0.0;
for(i=1;i≤m;i++)
   {
   if(w[i] > wmax)
      {
      wmax=w[i];
      }
   }
#ifdef DEBUG
printf("Secpmd Pass wmax=%f\n",wmax);
#endif DEBUG
wmin=wmax*(1.0e-6);
for(i=1;i≤m;i++)
   {
   if(w[i] < wmin)
      {
      w[i]=0.0;
      }
   }
svbksb(u,w,v,m,n,b,x);
#ifdef DEBUG
Print_Matrix("lhs",1,u,1,m,1,n);
Print_Vector("d",1,w,1,m);
Print_Matrix("rhs",1,v,1,m,1,n);
Print_Vector("Second pass solut",1,x,1,n);
nrdot(a,1,m,1,n,x,c);
Print_Vector("Second pass test solut b?)",1,c,1,m);
#endif DEBUG


a2=x[1];
c2=x[2];

free_matrix(a,1,m,1,n);
free_vector(b,1,m);
```

```
free_vector(c,1,m);
free_matrix(u,1,m,1,n);
free_matrix(v,1,m,1,n);
free_vector(w,1,n);
free_vector(x,1,n);
free_matrix(cvm,1,m,1,n);

#ifdef DEBUG
Debug=0;
#endif DEBUG

lcoeff[0][0]=a1;
lcoeff[0][1]=b1;
lcoeff[0][2]=0.0;
lcoeff[1][0]=a2;
lcoeff[1][1]=0.0;
lcoeff[1][2]=c2;
}
#endif OBSOLETE
/*****************************************************************************/
void NitherNoid(int mm,float **llist,float point[3],int debug_me)
{
/* Find the best point to solve a  mm long list of lines attached to llist.
 * Actually, the solution is defined the same as the best point to solve
 * for a list of lines consisting of pairs of planes.
 */
register int i,j,k;
float **u,*w,**v,**a,*b,*c,*x,**cvm;
float wmax,wmin;
int n=3; /* solution is in 3 dimensions */
int m=mm*2; /* each line consists of two planes */

#ifdef DEBUG
Debug=1;
#endif DEBUG

#ifdef DEBUG
if(debug_me)
for(j=0;j<mm;j++)
   {
   printf("N          :L  e[%d]:%+e,%+e,%+e\n\t%+e,%+e,%+e\n",j,
      *(llist[j]+0),*(llist[j]+1),*(llist[j]+2),
      *(llist[j]+3),*(llist[j]+4),*(llist[j]+5));
   }
```

```
#endif DEBUG

if(m < n )
   {
   fprintf(stderr,"Too few plane pa[illegible]   to solve for M<N\n");
   return;
   }

a=matrix(1,m,1,n);
b=vector(1,m);
c=vector(1,m);
u=matrix(1,m,1,n);
v=matrix(1,m,1,n);
cvm=matrix(1,m,1,n);
w=vector(1,m);
x=vector(1,n);

for(i=0;i<mm;i++)
   {
   b[i*2+1]=b[i*2+2]=1.0; /* point normal plane constant */

   u[i*2+1][1]=a[i*2+1][1]= *(llist[i]+0);
   u[i*2+1][2]=a[i*2+1][2]= *(llist[i]+1);
   u[i*2+1][3]=a[i*2+1][3]= *(llist[i]+2);

   u[i*2+2][1]=a[i*2+2][1]= *(llist[i]+3);
   u[i*2+2][2]=a[i*2+2][2]= *(llist[i]+4);
   u[i*2+2][3]=a[i*2+2][3]= *(llist[i]+5);
   }

#ifdef DEBUG
Print_Matrix("M[illegible]   put",1,a,1,m,1,n);
Print_Vector("M[illegible]   stants",1,b,1,m);
#endif DEBUG

svdcmp(u,m,n,w,v);

#ifdef SVD_EDIT
wmax=0.0;
for(i=1;i≤m;i++)
   {
   if(w[i] > wmax)
      {
      wmax=w[i];
```

```
      }
   }
wmin=wmax*(1.0e-6);
for(i=1;i≤m;i++)
   {
   if(w[i] < wmin)
      {
      w[i]=0.0;
      }
   }
#endif SVD_EDIT

svbksb(u,w,v,m,n,b,x);

#ifdef DEBUG
Print_Matrix("lhs",1,u,1,m,1,n);
Print_Vector("d"          ,1,w,1,m);
Print_Matrix("rhs",1,v,1,m,1,n);
Print_Vector("N"          solut"    ,1,x,1,n);
nrdot(a,1,m,1,n,x,c);
Print_Vector("N"          test solut"       ?)",1,c,1,m);
#endif DEBUG

point[0]=x[1];
point[1]=x[2];
point[2]=x[3];

free_matrix(a,1,m,1,n);
free_vector(b,1,m);
free_vector(c,1,m);
free_matrix(u,1,m,1,n);
free_matrix(v,1,m,1,n);
free_vector(w,1,n);
free_vector(x,1,n);
free_matrix(cvm,1,m,1,n);
}
/*****************************************************************************/
```

References

[1] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numer-*